CENG 3420
Computer Organization & Design

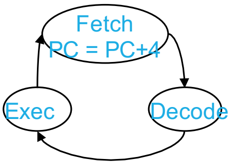
Lecture 09: Datapath

Bei Yu
CSE Department, CUHK
byu@cse.cuhk.edu.hk

(Textbook: Chapters 4.1 – 4.4)

Spring 2022

- We're ready to look at an implementation of RISC-V
- Simplified to contain only:
  - Memory-reference instructions: `lw, sw`
  - Arithmetic-logical instructions: `add, addu, sub, subu, and, or, xor, nor, slt, sltu`
  - Arithmetic-logical immediate instructions: `addi, addiu, andi, ori, xori, slti, sltiu`
  - Control flow instructions: `beq, j`
- Generic implementation:
  - Use the program counter (PC)
  - To supply the instruction address and fetch the instruction from memory (and update the PC)
  - Decode the instruction (and read registers)
  - Execute the instruction

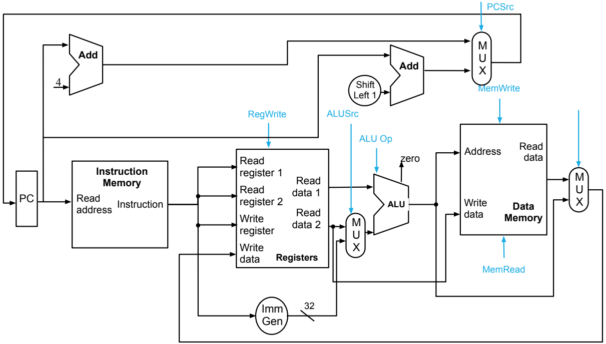


Note:

- memory reference: uses ALU to compute addresses
- arithmetic: uses the ALU to do the require arithmetic
- control: uses the ALU to compute branch conditions.

Abstract Implementation View

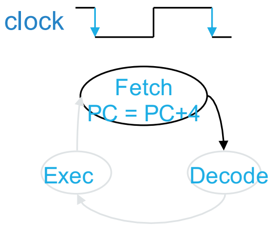
- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

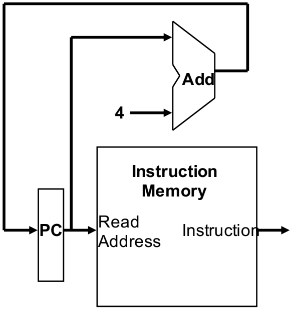


- Single cycle operation
- Split memory (Harvard) model - one memory for instructions and one for data

Fetching Instructions

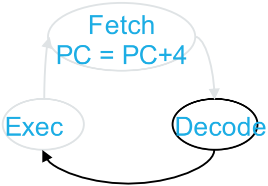
1. Reading the instruction from the Instruction Memory
2. Updating the PC value to be the address of the next (sequential) instruction
3. PC is updated every clock cycle, so it does not need an explicit write control signal
4. Instruction Memory is read every clock cycle, so it doesn't need an explicit read control signal

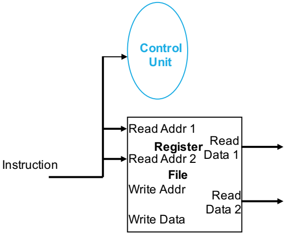




Decoding Instructions

1. Sending the fetched instruction's opcode and function field bits to the control unit
2. Reading two values from the Register File
3. (Register File addresses are contained in the instruction)





Reading Registers "Just in Case"

- Both RegFile read ports are active for all instructions during the Decode cycle
- Using the `rs1` and `rs2` instruction field addresses
- Since haven't decoded the instruction yet, don't know what the instruction is
- Just in case the instruction uses values from the RegFile do "work ahead" by reading the two source operands

Reading Registers "Just in Case"

- Both RegFile read ports are active for all instructions during the Decode cycle
- Using the `rs1` and `rs2` instruction field addresses
- Since haven't decoded the instruction yet, don't know what the instruction is
- Just in case the instruction uses values from the RegFile do "work ahead" by reading the two source operands

Question

Which instructions do make use of the RegFile values?

EX-1

All instructions (except `j`) use the ALU after reading the registers. Please analyze memory-reference, arithmetic, and control flow instructions.

EX-1

All instructions (except `j`) use the ALU after reading the registers. Please analyze memory-reference, arithmetic, and control flow instructions.

- Memory reference use ALU to compute addresses:
  `lw s1, 20 (s2)`

EX-1

All instructions (except `j`) use the ALU after reading the registers. Please analyze memory-reference, arithmetic, and control flow instructions.

- Memory reference use ALU to compute addresses:
  `lw s1, 20 (s2)`
- Arithmetic use the ALU to do the require arithmetic:
  `add s1, s2, s3 # (s1 = s2 + s3)`

EX-1

All instructions (except `j`) use the ALU after reading the registers. Please analyze memory-reference, arithmetic, and control flow instructions.

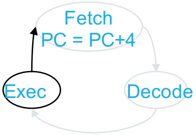
- Memory reference use ALU to compute addresses:
  `lw s1, 20 (s2)`
- Arithmetic use the ALU to do the require arithmetic:
  `add s1, s2, s3 # (s1 = s2 + s3)`
- Control use the ALU to compute branch conditions:
  `beq s1, s2, 25`

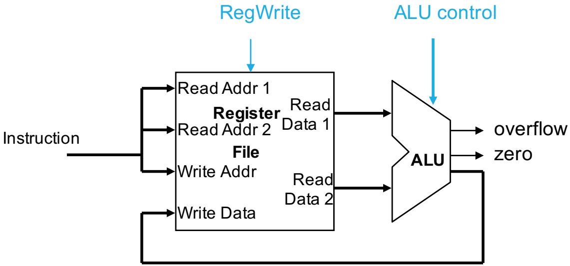
Executing R Format Operations

R format operations: `add, sub, sll, slt, xor, srl, sra, or, and`

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type

- Perform operation (`op`, `funct3` or `funct7`) on values in `rs1` and `rs2`
- Store the result back into the Register File (into location `rd`)
- Note that Register File is not written every cycle (e.g. `sw`), so we need an explicit write control signal for the Register File



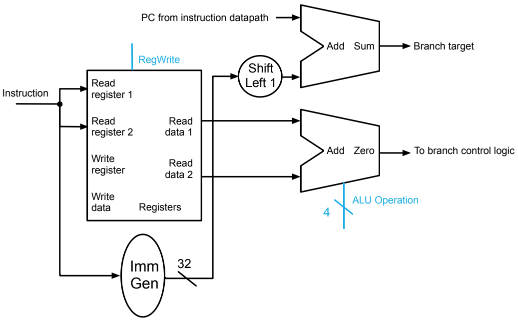


Consider the `slt` Instruction

- Remember the R format instruction `slt`

```
slt t0, s0, s1   # if    s0 < s1
                 # then  t0 = 1
                 # else  t0 = 0
```

- Where does the 1 (or 0) come from to store into `t0` in the Register File at the end of the execute cycle?



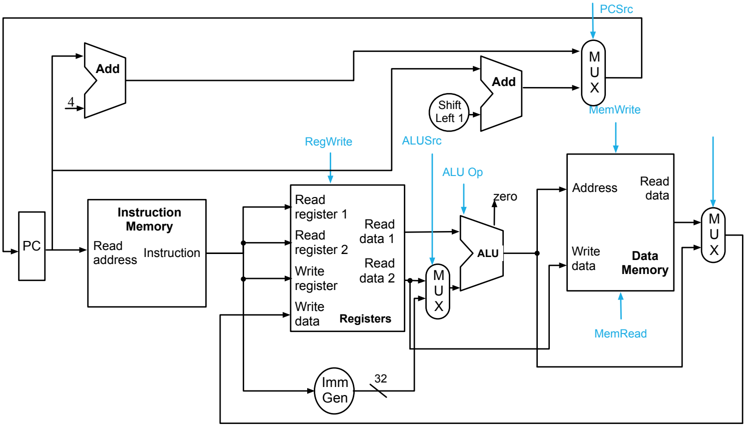
Executing Load and Store Operations

imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

Load and store operations have to

- compute a memory address by adding the base register (in `rs1`) to the 12-bit signed offset field in the instruction
  - base register was read from the Register File during decode
  - offset value in the low order 12 bits of the instruction must be sign extended to create a 32-bit signed value
- store value, read from the Register File during decode, must be written to the Data Memory
- load value, read from the Data Memory, must be stored in the Register File

Executing Load and Store Operations (cont.)



Executing Branch Operations

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

Branch operations have to

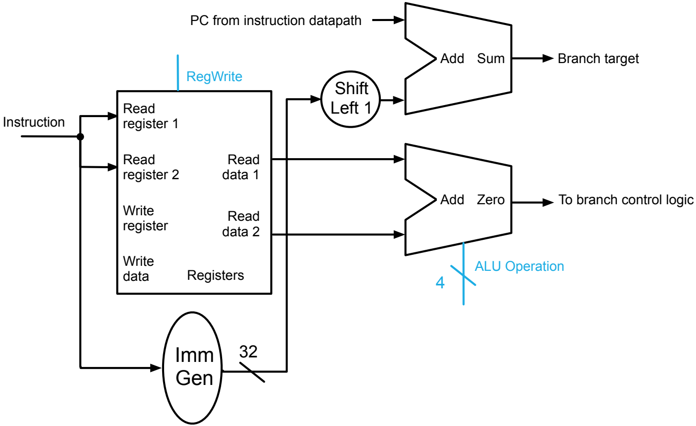
- compare the operands read from the Register File during decode (`rs1` and `rs2` values) for equality (zero ALU output)
- The 12-bit B-immediate encodes signed offsets in multiples of 2 bytes.
- The 12-bit immediate offset is sign-extended and added to the address of the branch instruction to give the target address.

imm[11:5]		rs2	rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

The only difference between S and B formats:

the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format

Executing Branch Operations (cont.)



Note:

- Textbook is about RV64: 64-bit memory space
- Our course is about RV32: 32-bit memory space
- RV32 is better suited to very low-cost processors
- Therefore, "Imm Gen" module outputs 32-bits.

Executing Jump Operations

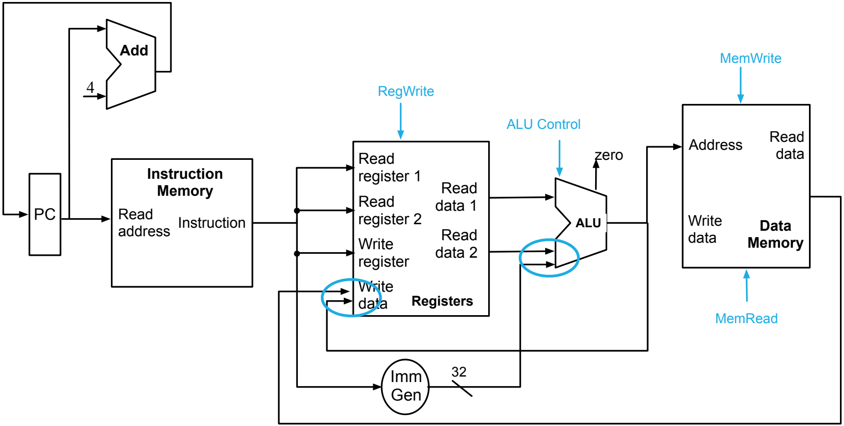
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode	J-type

- `jal`
- The J-immediate encodes a signed offset in multiples of 2 bytes.
- The offset is sign-extended and added to the address of the jump instruction to form the jump target address.

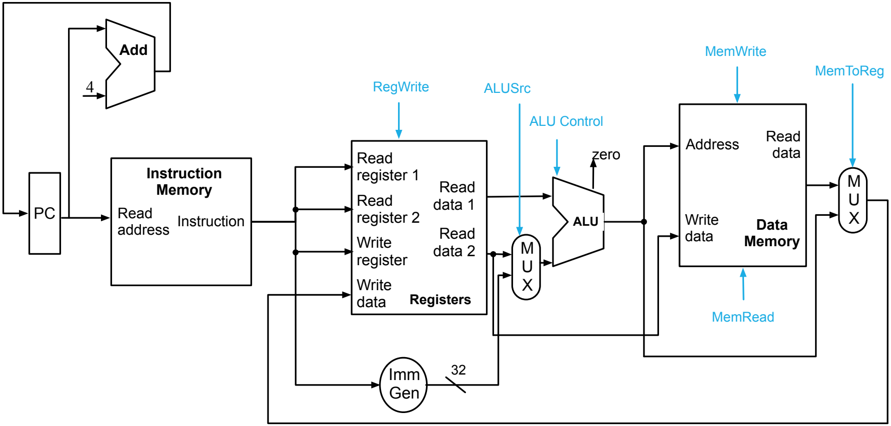
Creating a Single Datapath from the Parts

- Assemble the datapath elements, add control lines as needed, and design the control path
- Fetch, decode and execute each instruction in one clock cycle – single cycle design
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., why we have a separate Instruction Memory and Data Memory)
  - to share datapath elements between two different instruction classes will need multiplexors at the input of the shared elements with control lines to do the selection
- Cycle time is determined by length of the longest path

Multiplex Insertion



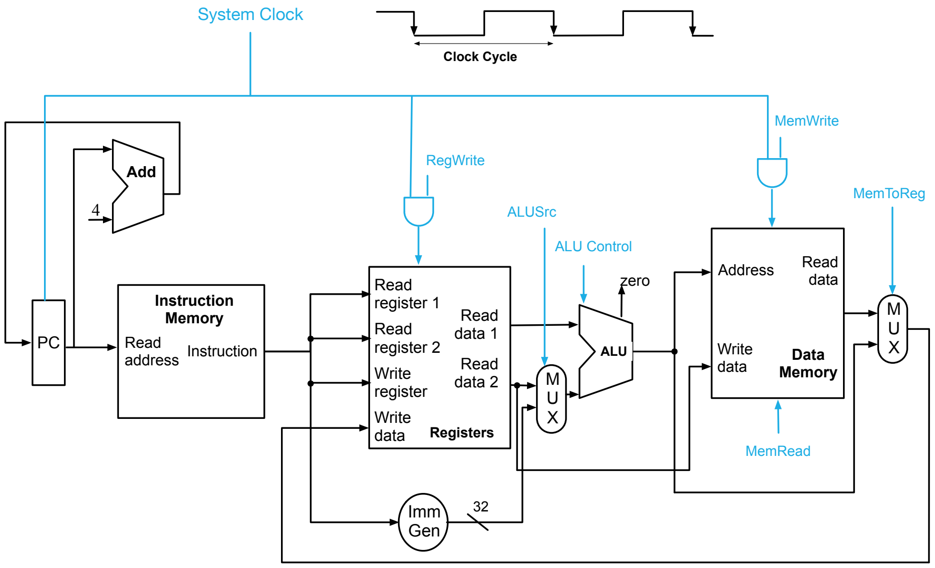
Multiplex Insertion



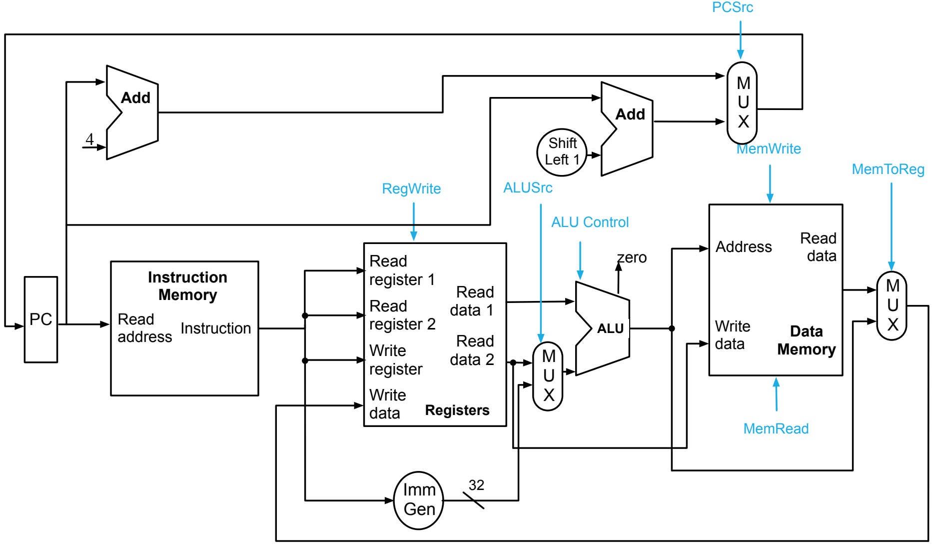
Note:

- `ALUSrc`: determine second ALU operand
- `MemtoReg`: whether feed memory data into register file

Clock Distribution



Adding the Branch Portion



Note:

- `PCSrc`: The PC is replaced by the output of the adder that computes the branch target.

Our Simple Control Structure

- We wait for everything to settle down
  - ALU might not produce "right answer" right away
  - Memory and RegFile reads are combinational (as are ALU, adders, muxes, shifter, signextender)
  - Use write signals along with the clock edge to determine when to write to the sequential elements (to the PC, to the Register File and to the Data Memory)
- The clock cycle time is determined by the logic delay through the longest path
- (We are ignoring some details like register setup and hold times)

Summary: Adding the Control

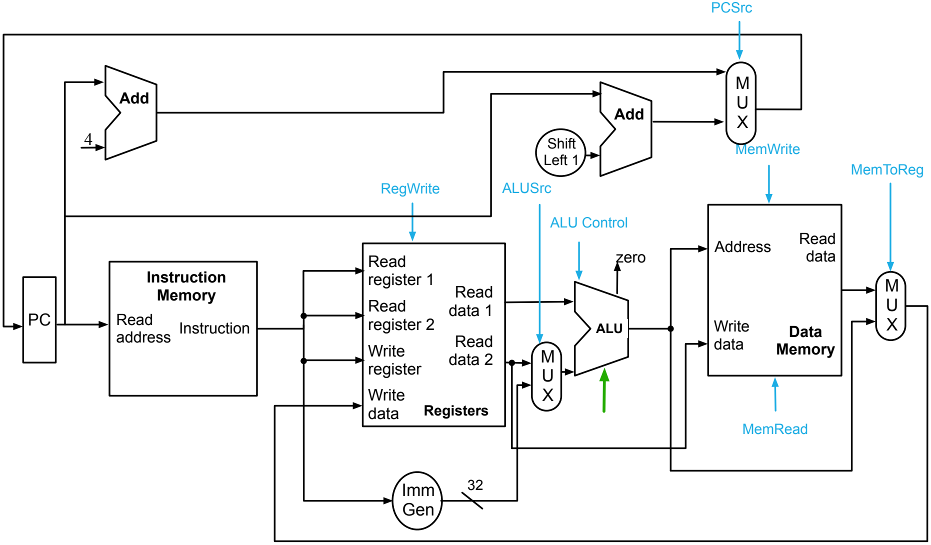
- Selecting the operations to perform (ALU, Register File and Memory read/write)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction

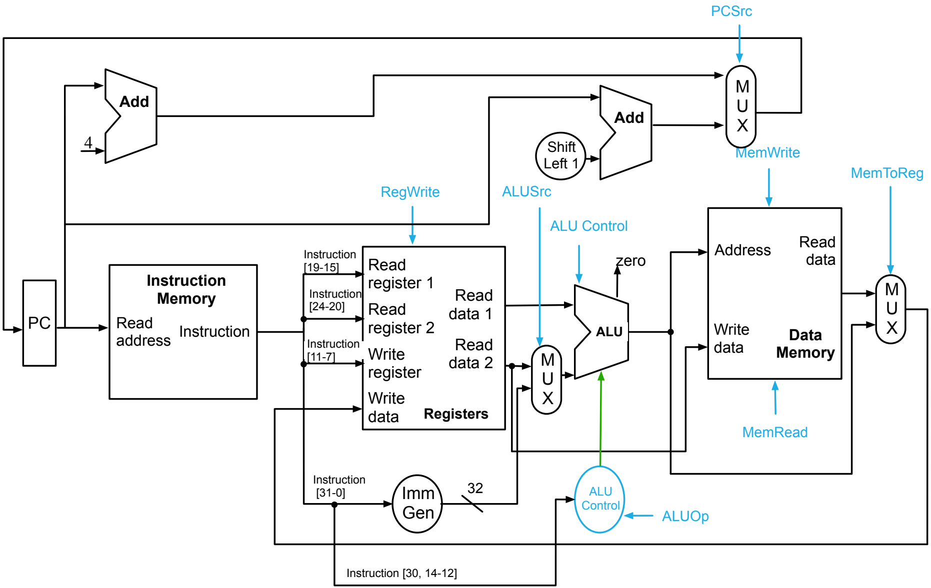
31–25	24–20	19–15	14–12	11–7	6–0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type

Observations:

- `opcode` field always in bits 6-0
- address of two registers to be read are always specified by the `rs1` and `rs2` fields (bits 19–15 and 24–20)
- base register for `lw` and `sw` always in `rs1` (bits 19–15)

(Almost) Complete Single Cycle Datapath




PCSrc
Add
4
M
U
X
Add
Shift Left 1
MemWrite
MemToReg
RegWrite
ALUSrc
ALU Control
zero
PC
Instruction Memory
Read address
Instruction
Instruction [19-15]
Instruction [24-20]
Instruction [11-7]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
ALU
Address
Read data
Write data
Data Memory
MemRead
Instruction [31-0]
Imm Gen
32
ALU Control
ALUOp
Instruction [30, 14-12]

ALU Control

ALU's operation based on instruction type and function code[1]

ALU control input	Function
0000	and
0001	or
0010	xor
0011	nor
0110	add
1110	subtract
1111	set on less than

[1] Notice that we are using different encodings than in the book

EX: ALU Control

Controlling the ALU uses of multiple decoding levels

- main control unit generates the `ALUOp` bits
- ALU control unit generates `ALUcontrol` bits

Instr op	funct	ALUOp	action	ALUcontrol
lw	xxxxxx	00		
sw	xxxxxx	00		
beq	xxxxxx	01		
add	100000	10	add	0110
subt	100010	10	subtract	1110
and	100100	10	and	0000
or	100101	10	or	0001
xor	100110	10	xor	0010
nor	100111	10	nor	0011
slt	101010	10	slt	1111

Instr op	funct	ALUOp	action	ALUcontrol
lw	xxxxxx	00	add	0110
sw	xxxxxx	00	add	0110
beq	xxxxxx	01	subtract	1110
add	100000	10	add	0110
subt	100010	10	subtract	1110
and	100100	10	and	0000
or	100101	10	or	0001
xor	100110	10	xor	0010
nor	100111	10	nor	0011
slt	101010	10	slt	1111

ALU Control Truth Table

F5	F4	F3	F2	F1	F0	ALU Op_1	ALU Op_0	ALU $control_3$	ALU $control_2$	ALU $control_1$	ALU $control_0$
X	X	X	X	X	X	0	0	0	1	1	0
X	X	X	X	X	X	0	1	1	1	1	0
X	X	0	0	0	0	1	0	0	1	1	0
X	X	0	0	1	0	1	0	1	1	1	0
X	X	0	1	0	0	1	0	0	0	0	0
X	X	0	1	0	1	1	0	0	0	0	1
X	X	0	1	1	0	1	0	0	0	1	0
X	X	0	1	1	1	1	0	0	0	1	1
X	X	1	0	1	0	1	0	1	1	1	1

ALU Control Truth Table

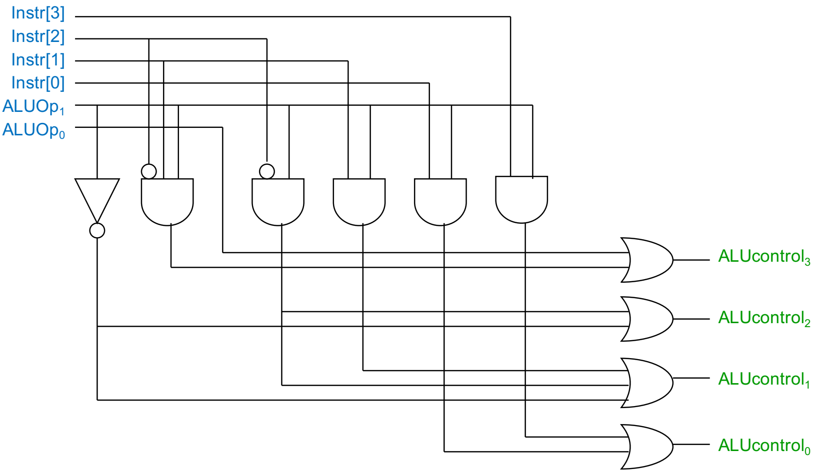
F5	F4	F3	F2	F1	F0	ALU Op_1	ALU Op_0	ALU $control_3$	ALU $control_2$	ALU $control_1$	ALU $control_0$
X	X	X	X	X	X	0	0	0	1	1	0
X	X	X	X	X	X	0	1	1	1	1	0
X	X	0	0	0	0	1	0	0	1	1	0
X	X	0	0	1	0	1	0	1	1	1	0
X	X	0	1	0	0	1	0	0	0	0	0
X	X	0	1	0	1	1	0	0	0	0	1
X	X	0	1	1	0	1	0	0	0	1	0
X	X	0	1	1	1	1	0	0	0	1	1
X	X	1	0	1	0	1	0	1	1	1	1

Add/subt

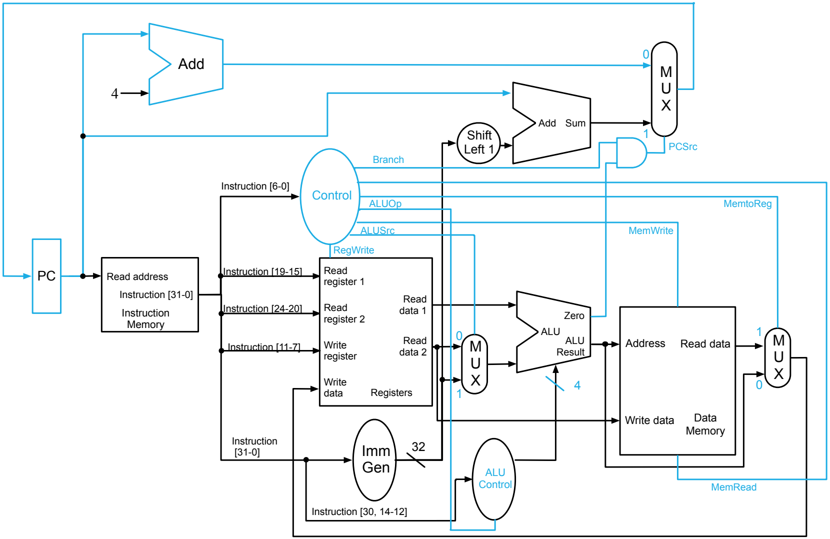
Mux control

ALU Control Logic

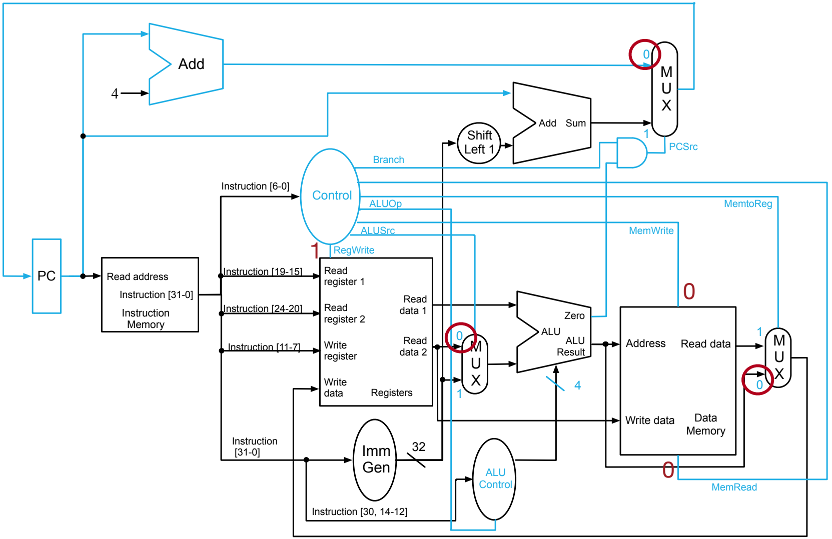
From the truth table can design the ALU Control logic



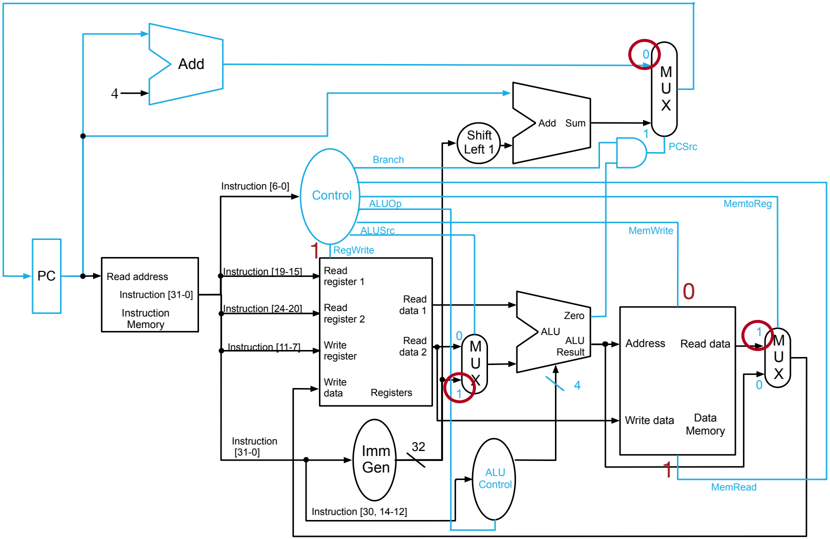
(Almost) Complete Datapath with Control Unit



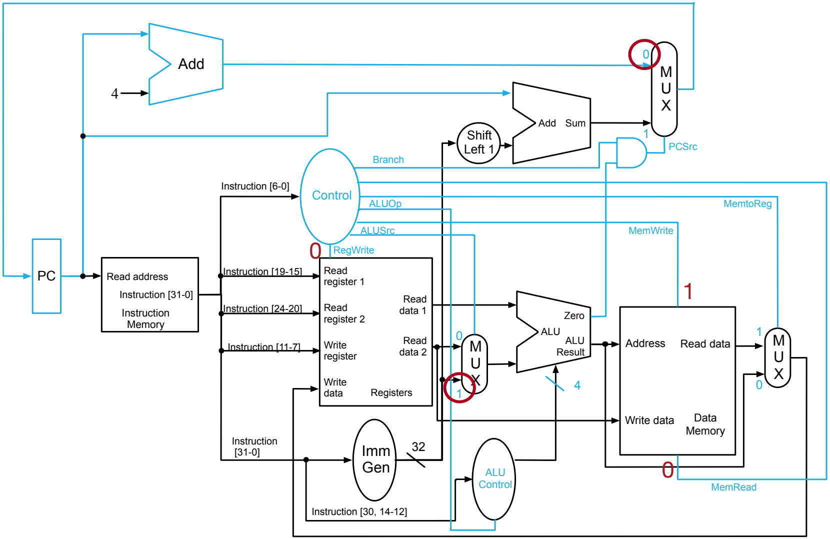
(Almost) Complete Datapath with Control Unit



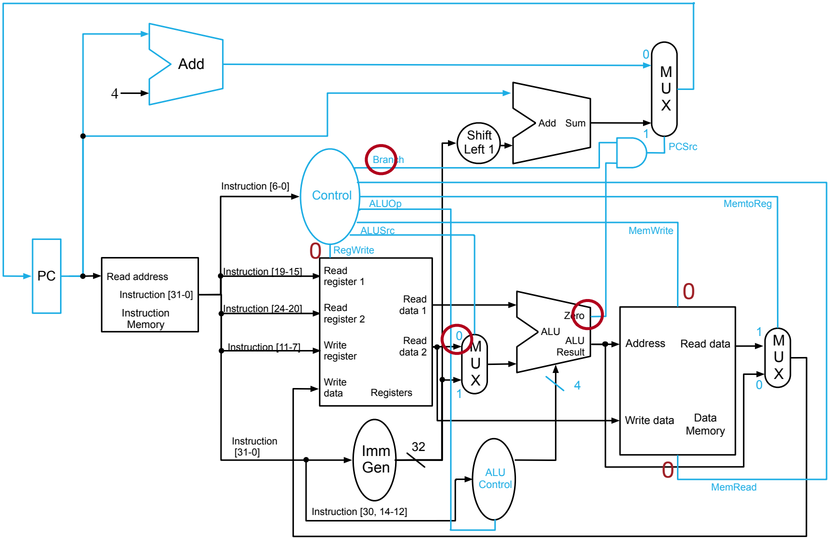
(Almost) Complete Datapath with Control Unit



(Almost) Complete Datapath with Control Unit



(Almost) Complete Datapath with Control Unit



Note:

Previous pictures gave control signales for instructions:

1. R-type
2. lw
3. sw
4. branch

Main Control Unit

Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp
R-type 000000	1	0	0	1	0	0	0	10
lw 100011	0	1	1	1	1	0	0	00
sw 101011	X	1	X	0	0	1	0	00
beq 000100	X	0	X	0	0	0	1	01

Control Unit Logic